Intel® Xeon Phi™ Processor Software

Optimization Guide

June 2016

Document Number: 334541-001

Legal Disclaimer

Contents

Tables

Revision History

Date	Revision	Description
June 2016	-001	Initial public release

§

1 Introduction

This document targets engineers interested in optimizing code for improved performance on the Intel® Xeon Phi™ processor. The manual begins with a high level description of the Intel® Xeon Phi™ processor micro-architecture. It follows with several topics that have the highest impact on performance on Intel® Xeon Phi™ AVX512 instructions, Memory Subsystems, Micro-architectural Nuances, Compiler Knobs & Directives, Numeric sequences, MCDRAM as Cache, and Scalar versus Vector Coding.

2 Basic Intel® Xeon Phi™ processor Microarchitecture Guide

Intel® Xeon Phi™ processor is generally broken into two parts, the tile and untile. Below, we discuss aspects related to them in more detail.

2.1 Tile

The Intel® Xeon Phi™ processor core is based on the Silvermont (SLM) micro-architecture. It has been modified to support AVX1, AVX2, AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512PF (prefetch), and AVX-512ER (exponential & reciprocal) ISA extensions. Each core supports up to 4 threads. Two cores share the same 1 MB L2, which comprises a tile. The number of enabled tiles varies based on SKU purchased. Tile frequency for the Intel® Xeon Phi™ processor varies, depending on workload, power budget, and SKU purchased.

Like SLM, up to 2 instructions per cycle can be decoded, allocated, and retired in a core. The frontend fetches bytes from the instruction cache, decodes instructions into uops, and predicts the destinations of branches. The Vector Processing Unit (VPU) performs up to two 512b FMAs per cycle. Floating point instructions from x87 are limited to 1 per cycle. The integer units can dispatch 2 uops per cycle each, and are able to do so out-of-order. The memory execution unit (MEC) dispatches 2 uops from its scheduler in-order, but uops can complete in any order. The data cache can read two 64B cache lines and write one cache line per cycle.

The frontend can fetch 16 bytes of instructions per cycle. The decoders can decode up to two instructions per cycle. The decoders can look at no more than 24 bytes in a cycle. The decoders can only provide a single uop per instruction. If more uops are required (like VSCATTER*) a performance bubble is taken to go to the MS (microcode sequencer) of 3-7 cycles, depending on instruction alignment in the decoder and length of the MS flow. The decoder will also have small delay if a taken branch is encountered. If an instruction has more than 3 prefixes, then there will be a multi-cycle bubble.

The reorder buffer (ROB) is 72 uops deep. There are 16 store buffers (for both address and data). The two integer schedulers (one per dispatch port) are 12 entries each. The single MEC scheduler has 12 entries, and dispatches up to 2 uops per cycle. The two VPU schedulers (one per dispatch port) are 20 entries each. The schedulers, ROB, and store data buffers are hard partitioned per thread (1, 2, or 4 thread mode). Hard partitioning of resources changes as threads wake up and go to sleep. The store address buffers have 2 entries reserved per thread, with the remaining entries shared among the threads.

The Intel® Xeon Phi™ processor is micro-architected to optimize bandwidth (BW) over latency. Loads to integer registers (e.g. RAX) are 4 cycles, and loads to VPU registers (e.g. XMM0, YMM1, ZMM2, or MM0) are 5 cycles. Only one integer load is possible per cycle, but the other memory operations (store address, vector load, and prefetch) can

dispatch two per cycle. Stores commit post-retirement, at a rate of 1 per cycle. The data cache and instruction caches are each 32 KB in size.

Most integer math operations have a BW of 2 and are single cycle. The Intel® Xeon Phi™ processor has a single integer multiplier with a latency of 3 or 5 cycles depending on operand size. VPU math operations are either 2 cycles (masks, integer vectors, Booleans, and 'simple' math) or 6 cycles (most math), and can dispatch on either VPU ports. The following instructions can only dispatch on a single port - x87 math operations, division, AVX-512ER, vector permute / shuffle, conflict instructions, AESni, and the data portion of a vector store. For those operations, they are limited to one of the two VPU dispatch pipes. Vector store data and vector to GPR moves are on one dispatch pipe. The remaining single pipe instructions are on the other dispatch pipe.

Table 1. Vector

Vector Instructions	Latency	Instructions / cycle
Simple integer	2	2
Most vector math (FMA)	6	2
Mask operations	2	2
X87 / MMX math	6	1
64b EMU (AVX-512ER)	7	0.5
32b EMU (AVX-512ER)	8	0.33
Shuffle / permutes (1 src)	2	1
Shuffle / permutes (2 src)	3	0.5
Convert – same width	2	1
Convert – different width	6	0.2
Vector Loads	5	2
Store to load forwarding	2	2
Gather (8 elements)	15	0.2
Gather (16 elements)	19	0.1
Vector to GPR move (MOVD xmm0 -> eax)	2	1
GPR to vector move (MOVD eax -> xmm0)	6	1
DIVSS or SQRTSS	25	~0.05
DIVSD or SQRTSD	40	~0.03
DIVPS, DIVPD, SQRTPS, or SQRTPD	38	0.1

Table 2. Integer scalar

Integer scalar instructions	Latency	Instructions / cycle
Most math	1	2
Integer multiply	3 or 5	1
Store to load forwarding	2	1
Integer Loads	4	1
Integer division	Variable	0.05

Table 3. Structure

Structure	Sets	Ways	Latency	Comments
uTLB	8	8	1	64 4k pages (fractured)
DTLB (4k)	32	8	4	256 4k pages
DTLB (2M/4M)	16	8	4	128 2M/4M pages
DTLB (1G)	1	16	4	16 1 GB pages
ITLB	1	48	4	48 4k pages (fractured)
PDE	8	4	1	Page descriptors
Data cache	64	8	4 or 5 cycles	32 KB
Instruction cache	64	8	4	32 KB
Shared L2 cache	1024	16	13 + L1 latency	1 MB

The Intel® Xeon Phi™ processor supports a compressed uop format in the frontend that allows some complicated instructions to be treated as a single uop in the frontend. Some of the encodings cannot be allocated with another uop, and we have a condition internally called "alloc by itself". These uops must be the only one allocated in that cycle. This is for uops that need to expand to two integer or vector uops. Examples of these instructions are POP (integer store data + ESP update), INC (ADD writing register and ADD writing partial flags), GATHER (2 VPU uops), and RET (JMP + ESP update). CALL, DEC, and 3 source LEA have similar limitations.

2.2 Untile

The Intel® Xeon Phi™ processor untile comprises a mesh connecting the tiles with MCDRAM and DDR memory. At each mesh stop, there is a connection to the tile and a tag directory that identifies which L2 cache (if any) holds a particular cache line. Unlike Xeon, there is no shared L3 cache. Memory accesses that miss in the tile must go over the mesh to the tag directory to identify any cached copies in another tile. Cache coherence uses the MESIF protocol. If the cache line is not cached in another tile, then a request goes to memory.

MCDRAM is an on-package high bandwidth memory that has individual peaks for read and write traffic that are lower than aggregate BW. DDR memory BW can potentially be saturated by writes or reads alone. The achievable memory BW for MCDRAM is approximately 4x - 6x of what DDR can do, depending on the mix of reads and writes.

MCDRAM has a capacity of 8 or 16 GB in Intel® Xeon Phi™ processor, depending on SKU (4 or 8 controllers). If less memory is installed, peak BW will decrease. Maximum DDR capacity is 384 GB. MCDRAM is higher BW and lower capacity than DDR.

Physical memory can be partitioned in various memory modes. This list is not complete, but a quick overview.

- MCDRAM as a direct mapped cache and DDR is backing memory (cache mode)
- MCDRAM and DDR as disjoint addressable memory (flat mode)
- MCDRAM partitioned so that some of it is a direct mapped cache, and part is directly addressable (hybrid mode)

The mapping of tag directories, tiles, and memory can also be controlled. Some of the clustering modes are:

- All to all: the requesting core, tag directory and memory controller for a line can be anywhere in the mesh.
- Quadrant: the tag directory and memory that it monitors are in the same quadrant of the mesh, but the requesting core can be anywhere in the mesh.
- SNC4: Also known as Sub-NUMA Clustering with 4 domains. Quadrant mode with the BIOS exposing multiple NUMA domains. Optimized SW can co-locate the requesting core, tag directory, and memory controller so that they are in the same quadrant of the mesh.

If critical portions of an application working set fit in the capacity of MCDRAM, performance could benefit greatly by allocating it into the MCDRAM and using flat or hybrid mode. Cache mode is generally best for code that has not yet been optimized for the Intel® Xeon Phi™ processor, and has a working set that MCDRAM can cache.

Quadrant mode is the default mesh configuration. SNC4 clustering requires some support from SW to recognize the different NUMA nodes. If DDR is not populated evenly, e.g. missing DIMMs, the mesh will need to use the all to all clustering mode.

3 AVX-512 vs. AVX2

Some basic notes for Intel® Xeon Phi™ processor and AVX-512 support to keep in mind are following.

Intel® Xeon Phi™ processor performs well with accesses that cross cache lines, but is quicker if loads and stores do not cross cache lines. It is beneficial for performance if a programmer aligns vectorized arrays properly. It is not required for correctness.

AVX-512F supports 512b vector registers that are laid on top of the existing XMM [0..127] and YMM [0..255] registers. Using AVX-512 instruction encodings, a programmer can access up to 32 ZMM registers.

AVX-512F defines a set of 8 mask registers (k0-k7). A mask register is a bit mask corresponding to each element of the vector (512b / 32b or 64b elements yields 16 or 8 bit wide masks). The mask register can be used in mask only operations (KAND), the target of a vector operation (VCMPPS), or as a write mask for 'normal' vector operations (VMULPD). For write masks, the destination will be the result of the operation specified by the instruction if the corresponding mask bit is set. If the corresponding mask bit is not set, then the value written in the destination will either be the destination value (merge) or '0 (zeroing). Instructions that use merging have an implicit dependency on the destination register, zeroing does not, which might change the dependency graph. Merging or zeroing is specified by the instruction. If k0 is used as the write mask, the instruction acts as if the operation is un-masked (all mask bits are set).

Read the AVX-512 documentation for details on per instruction rounding modes and SAE (suppress all exceptions).

AVX-512F memory accesses support the memory fault suppression feature. If a memory access would have generated a memory fault (segmentation, page, or canonical), the write mask is examined. Faults are suppressed if no faulting memory locations are accessed based on the write mask. For instance, a memory access with a write mask of '0 will never generate a memory fault. This behavior makes all AVX-512 instructions that touch memory have similar faulting semantics to VMASKMOV. If a potential memory fault (e.g. page fault) is detected, there is a several dozen cycle penalty to confirm that the fault is real, or to suppress it.

The Intel® Xeon Phi™ processor supports the PREFETCHW and PREFETCHWT1 instructions. These are software prefetches with the ownership hint ('E' state) to the L1 and L2 cache, respectively. This software prefetch should be used if the programmer knows that the cacheline is likely to be written to soon.

3.1 AVX2 and previous Intel® Xeon Phi™ processors

The Intel® Xeon Phi™ processor fully supports all ISA from Broadwell, except the TSX instruction set. In addition, the Intel® Xeon Phi™ processor supports AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512PF (prefetching), and AVX-512ER (exponential and reciprocal) ISA extensions. With these new instructions, Intel® Xeon Phi™ processor supports the ZMM register set (32 512b vector registers – ZMM0-31) and mask registers (8 64b mask registers – k0-7).

Intel® Xeon Phi™ coprocessor, the previous generation Intel Phi processor, does not support the AVX-512 ISA, but a similar 512b vector ISA. Many of the instructions in Intel® Xeon Phi™ coprocessor and AVX-512 are similar, with different encodings. However, some instructions and options are very different. Intel® Xeon Phi™ coprocessor behavior for scatters and gathers is vastly different than AVX-512. Experienced Xeon Phi assembly programmers should carefully review the AVX-512 ISA. There are no other announced processors that support the Intel® Xeon Phi™ coprocessor ISA extensions.

3.2 Gather and scatter instructions

Intel® Xeon Phi™ processor supports scatter and gather instructions. Gather can be thought of as multiple element size loads, where the elements are merged into a single vector register. A second vector register is used to specify multiple addresses. A scatter is comprised of multiple element sized stores, where one vector register holds the data to be stored, and another vector register specifies the multiple addresses for the stores. AVX2 supported gathers for XMM and YMM vectors, but does not support scatter.

AVX-512F versions of gather and scatter are quite different from the Intel® Xeon Phi™ coprocessor implementation, which would only process a single cacheline on each instruction. The AVX-512F version of the instructions complete the entire gather or scatter in a single instruction. There are several micro-architectural improvements that allow Intel(R) Xeon Phi(TM) processor to be faster than Broadwell for AVX2 gathers.

Gather and Scatter support various index, element, and vector widths. The AVX-512 flavors of gather and scatter use the mask registers to identify the lanes that should be loaded to or stored from the vector registers. AVX2 does not support scatter instructions or mask registers. In addition, AVX2 is limited to at most a 32B vector width. Intel® Xeon Phi™ coprocessor supports scatter and gather instructions with mask registers and 64B vector registers. For correct completion of a Intel® Xeon Phi™ coprocessor gather or scatter, the programmer would need to wrap the instruction in a test of the mask for 0, and a conditional branch.

For instance, if you wrote the following C code fragment:
```
for (uint32 i=0; i < 16; i++)
     b[i] = a[indirect[i]];
```

The equivalent AVX2 code would be something like this:
```
vmovdqu      ymm0, [rsp+0x1000]  ;; load indirect[]
vmovdqu      ymm3, [rsp+0x1020]  ;;   part 2
vpcmpeqd     ymm4, ymm4, ymm4    ;; setup mask
vmovdqa      ymm1, ymm4          ;;   part 2
vpgatherdd   ymm2, [rax+ymm0*4], ymm1
vpgatherdd   ymm5, [rax+ymm3*4], ymm4
vmovdqu      [rsp], ymm2         ;; store b[]
vmovdqu      [rsp+0x20], ymm5    ;;   part 2
```

The equivalent Intel® Xeon Phi™ coprocessor code would be something like this:

```
vmovaps      zmm0,[rsp+0x1000]   ;; load indirect[]
kxnor        k1,k1               ;; setup mask for gather
gather_loop:
vgatherdps   zmm2{k1},[rax+zmm0*4]
vgatherdps   zmm2{k1},[rax+zmm0*4]
jknzd        k1,<gather_loop>    ;; see if gather is done
vmovaps      [rsp],zmm2          ;; store b[]
```

The equivalent AVX-512F code would be something like this:

```
vmovups      zmm0, [rsp+0x1040]     ;; load indirect[]
kxnor        k1, k0, k0             ;; mask for gather
vpgatherdd   zmm1{k1}, [rsi+zmm0*4] ;; gather a[]
vmovdqu32    [rsp], zmm1            ;; store b[]
```

The AVX-512F code is more compact than the other instruction formats. In addition to using fewer instructions, the code can be executed faster than the alternate ISA sequences.

Intel® Xeon Phi™ processor internally implements GATHER* and SCATTER* using a scatter/gather index table (SGIT) and finite state machine (FSM) to generate the needed memory operations. The SGIT is 4 entries deep and hard partitioned by thread. Entries are allocated in the AIR, and de-allocated when all memory operations connected to the entry have exited the MEC. Each entry holds the base address, the array of indices, the mask value, and some auxiliary info on the operation being done. The FSM will insert up to 2 element sized memory operations into the MEC every cycle, taking (number of elements)/2 cycles to complete insertion for an instruction. The FSM will insert memory operations, regardless of the associated mask value. When the FSM finishes with one SGIT entry, it will start on the next, until there are no un-serviced SGIT entries.

3.3 Reciprocals, prefetch, & conflict

These relatively small sets of instructions are important for efficient vectorization of HPC programs. A short summary of the instructions is below:

CPUID	Instructions	Description
AVX512-PF	PREFETCHWT1	Prefetch cache line into the L2 cache with intent to write
	VGATHERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache
	VSCATTERPF{D,Q}{0,1}PS	Prefetch vector of D/Qword indexes into the L1/L2 cache with intent to write
AVX512-ER	VEXP2{PS,PD}	Computes approximation of 2^x with maximum relative error of 2^{-23}
	VRCP28{PS,PD}	Computes approximation of reciprocal with max relative error of 2^{-28} before rounding
	VRSQRT28{PS,PD}	Computes approximation of reciprocal square root with max relative error of 2^{-28} before rounding
AVX512-CD	VPCONFLICT{D,Q}	Detect duplicate values within a vector and create conflict-free subsets
	VPLZCNT{D,Q}	Count the number of leading zero bits in each element
	VPBROADCASTM{B2Q,W2D}	Broadcast vector mask into vector elements

The AVX512-PF instructions are used to aid memory performance, when the programmer or compiler knows with a high degree of certainty the set of cachelines that will be accessed in the near future. Their relationship to gather and scatter is similar to the relationship of prefetch(w)* to loads and stores. They should be used in equivalent scenarios, where the compiler or SW writer is comfortable in providing hints to HW on which lines should be in cache. Prefetch instructions do not impact architectural state.

The AVX-512ER instructions provide high precision approximations of exponential, reciprocal, and reciprocal square root functions. Approximations from earlier ISA extensions, like rcp11ps are far less accurate. An accurate approximation can reduce execution time for iterative algorithms like Newton-Raphson. Below is sample code using the Newton-Raphson algorithm to compute a single 32b float division with vrcp28ss. Both values are read off the stack. Note the use of rounding mode overrides on some of the math operations.

```
vgetmantss      xmm18, xmm18, [rsp+0x10], 0
vgetmantss      xmm20, xmm20, [rsp+0x8], 0
vrcp28ss        xmm19, xmm18, xmm18
vgetexpss       xmm16, xmm16, [rsp+0x8]
vgetexpss       xmm17, xmm17, [rsp+0x10]
vsubss          xmm22, xmm16, xmm17
vmulss          xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss    xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss     xmm21, xmm19, xmm20
vscalefss       xmm0, xmm21, xmm22
```

The AVX-512CD instructions allow for efficient vectorization of several access patterns. A common scheme can be characterized as the “histogram update”. This is when a memory location is read, operated on, and then stored to. A sample piece of C code with this access pattern:

```
for (i=0; i < 512; i++)
     histo[key[i]] += 1;
```

This code can be incorrectly vectorized with a gather, vpadd, and a scatter. To get the correct answer with vectorization, you must worry about cases where key[n] and key[m] have the same value, and n and m are in the same vector chunk. The AVX-512CD instructions detect these cases, and permit correct vectorization of the loop.

Below is the main loop disassembly from a compiler that vectorizes the histogram update code listed before:

```
Top:
 vmovups        zmm4, [rsp+rdx*4+0x40]
 vpxord         zmm1, zmm1, zmm1
 kmovw          k2, k1
 vpconflictd    zmm2, zmm4
 vpgatherdd     zmm1{k2}, [rax+zmm4*4]
 vptestmd       k0, zmm2, [rip+0x185c]
 kmovw          ecx, k0
 vpaddd         zmm3, zmm1, zmm0
 test           ecx, ecx
 jz             <No_conflicts>
 vmovups        zmm1, [rip+0x1884]
 vptestmd       k0, zmm2, [rip+0x18ba]
 vplzcntd       zmm5, zmm2
 xor            bl, bl
 kmovw          ecx, k0
 vpsubd         zmm1, zmm1, zmm5
Resolve_conflicts:
 vpbroadcastd   zmm5, ecx
 kmovw          k2, ecx
 vpermd         zmm3{k2}, zmm1, zmm3
 vpaddd         zmm3{k2}, zmm3, zmm0
 vptestmd       k0{k2}, zmm5, zmm2
 kmovw          esi, k0
 and            ecx, esi
 jz             <No_conflicts>
 add            bl, 0x1
 cmp            bl, 0x10
 jb             <Resolve_conflicts>
No_conflicts:
 kmovw          k2, k1
 vpscatterdd    [rax+zmm4*4]{k2}, zmm3
 add            edx, 0x10
 cmp            edx, 0x400
 jb             <Top>
```

4 Total number of threads

An individual thread has the highest performance when running as the only thread on a core. As thread count per core grows to 2 or 4, some applications will have higher per core performance, but lower per thread performance. If an application can perfectly scale its performance to an arbitrary number of threads, 4 threads per core is likely to have the highest instruction throughput. Practical limitation on memory capacity or parallelism may limit the number of threads per core.

Intel® Xeon Phi™ processor partitions many per core resources in fourths when using 3 or 4 threads on a core – like the ROB or scheduler. Because of this, a 3 thread configuration will have fewer aggregate resources than 1, 2, or 4 threads per core. Placing 3 threads on a core is unlikely to perform better than 2 or 4 threads per core.

5 Memory subsystem

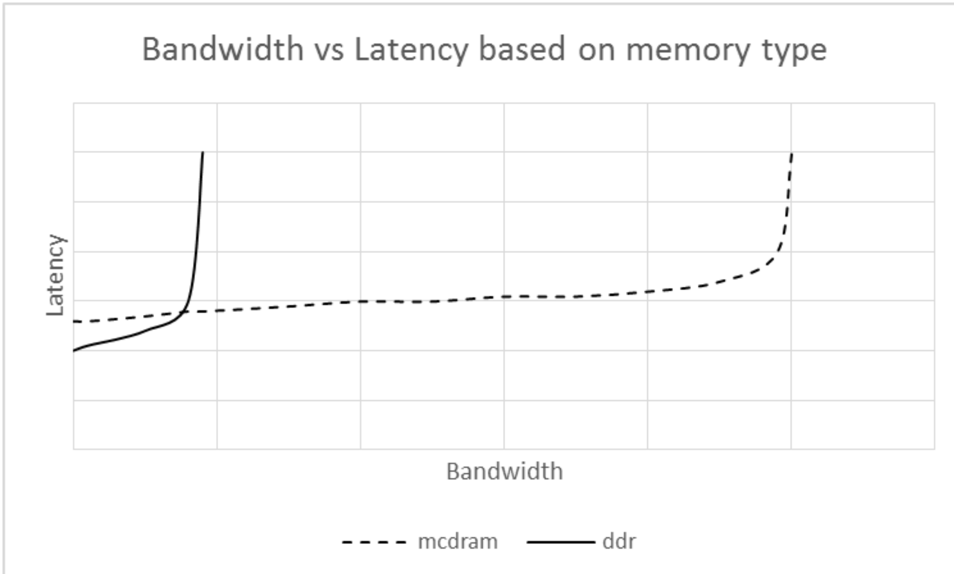
5.1 Caches

Intel® Xeon Phi™ processor has 32 KB caches for instructions and data. These are commonly referred to as the "level 1" (L1) caches. The two cores in a tile share a 1 MB cache that can hold a mix of instructions and data. These are commonly referred to as the "level 2" (L2) caches. When multiple tiles read the same cache line, each tile might have a copy of the cache line. If both cores in the same tile read a cache line, there will only be a single copy in the L2 cache of that tile. Cache lines found in the L1 cache can be accessed with high BW and low latency. Cache lines found in the L2 cache have lower BW and higher latency than the L1 cache, but higher BW and lower latency than an access to memory.

If MCDRAM is configured as a cache, then it can hold data or instructions accessed by the cores in a single place. If multiple tiles request the same line, only one MCDRAM cacheline will be used.

5.2 MCDRAM and DDR

MCDRAM and DDR memory have different latency and throughput profiles. When using the various memory modes (cache / flat) and deciding on where to allocate memory, this knowledge is important. In most memory configurations, the DDR capacity will be substantially larger than MCDRAM capacity. Likewise, MCDRAM capacity should be much larger than the combined L2 cache. Working sets that fit in MCDRAM capacity, but not in the L2 caches should be in MCDRAM. Large or rarely accessed structures should migrate to DDR. The Intel® Xeon Phi™ processor HW will try to do this dynamically if MCDRAM is put in cache or hybrid memory modes. If memory is in the flat memory mode, data structures are bound to one memory or the other (MCDRAM or DDR) at allocation time. The programmer should strive to maximize the number of memory access that go to MCDRAM. One possible algorithm would allocate data structures into MCDRAM if they are frequently accessed, and have working sets that do not fit into the tile caches.



In cache memory mode, the MCDRAM access is done first. If the cacheline is not in MCDRAM, then the DDR access begins. Because of this, the perceived memory access latency of DDR in cache memory mode is higher than in flat memory mode.

6 Micro-architecture Nuances (tile)

In this section, we review several suggestions on how to optimize the performance of the Intel® Xeon Phi™ processor tile.

6.1 Frontend

The Intel® Xeon Phi™ processor frontend (instruction cache, decoders, & branch prediction) handles most HPC code well. There are some restrictions that rarely impact performance that optimizers and compiler writers should be aware of.

The frontend can fetch 16 bytes per cycle. Since the Intel® Xeon Phi™ processor can process up to 2 instructions per cycle, this could limit performance when the average instruction length is greater than 8 bytes. The only common HPC instructions that use more than 8 bytes are vector math instructions with a memory source that has a 4 byte displacement. If trying to execute a long sequence of such operations, constraining the displacement size will provide an improvement in performance.

When selecting instructions, try to avoid instructions with many prefixes. If an instruction has more than 3 prefixes, the hardware requires more time to determine instruction boundaries. The penalty will happen each time the instruction is encountered. This is generally not an issue, but some instructions, like ADCX and ADOX with the REX prefix, are affected.

The branch predictors assume that a branch destination is in the same 4GB as the source (virtual). Far branches to dynamically linked libraries (DLLs) or server code that is not 'near' the source branch will be predicted poorly. Try to limit OS calls and the frequency that DLLs that are accessed. When possible, statically linking libraries can help. This is especially important for short latency library calls that are accessed frequently – whether MPI, a math function, or a network driver. If frequently accessed DLLs are preferred, experts can experiment with keeping the virtual addresses of user and dynamic library code near. One option is to use glibc 2.23 or later, with LD_PREFER_MAP_32BIT_EXEC.

6.2 Integer

If you need to PUSH or POP several values sequentially, it is better to issue multiple stores or loads (each using a unique RSP offset), and then update RSP once. Each PUSH/POP instruction generates its own uop to update RSP. This puts more pressure on allocation and integer execution bandwidth, just like INC and DEC. PUSH and POP instructions are commonly used as part of the calling convention, and can cause some bubbles if there are many function calls.

The Intel® Xeon Phi™ processor does not optimize the performance of 8b and 16b integer partial registers (like AH, AL, & AX). For the Intel® Xeon Phi™ processor, each reference to an integer register reads and writes the full 64b version. If you are using code that was written when 32b CPUs were a new concept, this might be an issue, but is unlikely to be a problem for high performance applications. It is simplest, and

generally faster, to just refer to the 32b and 64b versions of the integer registers (like EAX & RAX).

The Intel® Xeon Phi™ processor does not optimize partial flag writes. Instructions that write EFLAGs status bits will write all the status bits. If a subset of the bits is unchanged by an instruction, then the EFLAGs writer cannot execute until the previous writer of EFLAGs has completed. This can cause unexpected performance drops relative to recent Core microprocessors. INC and DEC are the two most frequently encountered instructions that are affected by this. A compiler should use "SUB 0x1" or "ADD 0x1" instead.

Integer division is a common mathematical operation, used whenever two integers are divided, or a modulus operation is specified. Unfortunately for those who care about performance, computing this operation is quite slow. If the integers are known to be relatively small (16 bits or less), there are fast SW sequences to emulate the division. If the divisor is known to be a power of 2, please use SHR (division) and/or AND (remainder) instead of DIV. Division by a constant can be replaced by MUL with a constant. If the input values are highly constrained, a pre-computed lookup table is likely to provide better performance. Division instructions should be aggressively minimized by the compiler, either using the techniques mentioned earlier, or by hoisting redundant divisions out of inner loops.

6.3 Vector

Minimize usage of x87, MMX, and SSE instructions. Use the AVX (packed and scalar) equivalents (128b, 256b, or 512b width) as much as possible. SSE and x87 instructions are generally similar performance to AVX, but are occasionally much slower than the AVX* equivalents. There are potential performance glass jaws when mixing SSE with 256b or 512b operations mentioned later on. If needed, MMX and x87 code can be freely interleaved with AVX code.

One class of instructions that are slow on the Intel® Xeon Phi™ processor are COMIS* and UCOMIS*. Replacing these instructions with the two instruction sequence of VCMPS* and KORTEST is almost always better for performance.

Some instructions, like VCOMPRESS*, are single uop when writing a register, but an MS flow when writing memory. Where possible, it is much better to do a VCOMPRESS to register and then store it. Similar optimizations apply to all vector instructions that do some sort of operation followed by a store (e.g. PEXTRACT).

MASKMOVQ and similar instructions should be replaced by k-masked ZMM load/store instructions.

The Intel® Xeon Phi™ processor does not have the same restrictions on mixing SSE and AVX code that Xeon processors have. Because of this, insertion of VZEROUPPER* is almost always a mistake for the Intel® Xeon Phi™ processor. The Intel® Xeon Phi™ processor will suffer some performance bubbles if an AVX instruction that operates on more than 128b is allocated before all previous SSE instructions have retired. This is a one-time bubble, with a performance loss comparable to a mispredicted branch. When compiling for the Intel® Xeon Phi™ processor, it is best to not generate SSE instructions. If the binary has been freshly compiled, only legacy libraries should have SSE code in them. This limits any performance hiccups to a few instructions after returning from legacy library calls. Extremely bad code generation for the Intel® Xeon Phi™ processor

would alternate SSE instructions with wider AVX instructions. If you wish to achieve excellent Intel® Xeon Phi™ processor performance, avoid this.

The Intel® Xeon Phi™ processor does not optimize the performance of VZEROUPPER or VZEROALL. The instruction requires about 10 cycles of BW in the allocation, execution, and retirement pipelines. The frontend of the Intel® Xeon Phi™ processor uses the microcode sequences (MS) for VZEROUPPER, and will need about 15 cycles. Xeon processors generally take a cycle of bandwidth at each point for VZEROUPPER. The Intel® Xeon Phi™ processor code should avoid both instructions as much as possible.

Experience has shown that intrinsic heavy code from earlier Xeon Phi implementations is unlikely to generate optimal Intel® Xeon Phi™ processor code. Intel® Xeon Phi™ coprocessor has free simple permutes and a more limited ISA relative to the Intel® Xeon Phi™ processor. An expert coder will want to scrub legacy Intel® Xeon Phi™ intrinsics to limit the number of MS flows generated, and to fix choices imposed by Intel® Xeon Phi™ coprocessor limitations (unaligned memory accesses). Large speedups are possible from even a quick review of legacy intrinsics. It is likely that coding the algorithm in a high level language (C / Fortran) and compiling will generate faster code than using Intel® Xeon Phi™ coprocessor intrinsics.

The HADDP* instruction is not the best way to do a horizontal addition on the Intel® Xeon Phi™ processor. The instruction does not have an AVX-512 version, and is limited to narrower width vectors. There are examples later in the document that list efficient methods to do horizontal reductions.

Some vector math instructions require multiple uops to implement in the VPU. This increases the latency of the individual instruction beyond the standard math latencies of 2 and 6. In general, instructions that alter element width (e.g. VCVTSD2SI) fall into this category. Many AVX2 instructions that operate on byte and word quantities have reduced performance compared to the equivalents that operate on 32b or 64b quantities.

The VPU is relatively large, and some logic blocks are distant from others. When a value must move between the two, due to a uop reading a register that another uop wrote, a delay of up to 2 cycles might be inserted to allow the value to arrive. There is no delay between uops that execute in the same block (FMADDPD -> MULPD -> XORPD) which is the vast majority of cases. The blocks that implement VPU Shuffles, x87 math, and byte/word multiplies are distant from the other blocks. Code that frequently transitions between logic blocks that are distant, will see longer dependence chains in the VPU.

There are a few instructions relevant to HPC that are long latency. The most commonly used are FP division, FP square root, and integer division. The programmer and/or compiler should carefully consider various tradeoffs for each of these instructions. The DIVPD, DIVPS, SQRTPD, and SQRTPS instructions are implemented as MS flows that use Newton-Raphson to iterate to an IEEE correct answer. The scalar versions of these instructions are shorter MS flows that use a HW unit. The integer divide performance is similar to Atom cores, and can be surprisingly slow to programmers.

Compiler options like '-no-prec-div' in the Intel compilers take advantage of loosening IEEE compatibility requirements, and can generate faster instructions sequences with the AVX-512ER instructions.

For FP divisions and square roots, it is greatly beneficial to be able to vectorize. The vector versions are approximately as fast as the scalar versions, but being able to do 8 or 16 operations in the same vector greatly improves performance.

Many complex math operations (with real and imaginary values) benefit from the VFMADDSUB* and VFMSUBADD* instructions. Please use them when appropriate.

The vector execution units of the Intel® Xeon Phi™ processor can consume a large portion of the power budget. Because of this, vector execution rate is monitored. If the core tries to issue vector uops quickly over a sustained period of time, a request will be sent to the power control unit (PCU) for a license to issue vector uops more frequently. In order for the PCU to grant the license, it may need to increase core voltage or reduce the ability to increase tile frequency (turbo). Until the license is granted, vector issue rate will be constrained to remain within power limits.

AVX-512 permits instructions to use masking. The best performance usually happens for instructions that do not use masking. Instructions that mask with zeroing usually have similar performance. Instructions that mask with merging can occasionally perform much slower, since there is an additional dependency introduced to the instruction. Keep in mind performance considerations when selecting the masking mode.

6.4 Memory

The Intel® Xeon Phi™ processor performs well on accesses that split cache lines, but there is still a performance penalty relative to accesses that do not split cache lines. If an algorithm has an access pattern that streams through memory, it would be beneficial to align as many of the accesses as possible to a 64B boundary. Likewise, if you want a 32B value (YMM), do not access 64B (ZMM) in memory and then mask off the last 32B. This may create cacheline splits for no reason.

The Intel® Xeon Phi™ processor is cautious when a load or store access crosses a 4 KB boundary. This happens whether or not the page being accessed is 4 KB in size or larger. This introduces a larger performance penalty for the access that crosses the boundary. If an algorithm is streaming through memory with 64B loads that are not aligned to a 64B boundary, every 64th load will cross the 4 KB boundary. It is best to align the access pattern as much as possible. If it is not possible, consider inserting a few SW prefetches to the L2 cache (PREFETCHT1 and similar instructions) several iterations ahead of the stream. This will start the page translation early and permit the L2 HW prefetcher to start fetching the stream on the next page.

Some access patterns for gather and scatter will always have pairs of consecutive addresses. One common example are complex numbers, where the real and imaginary parts are laid out contiguously. It is also common when w, x, y, and z information is contiguous. If the values are 32b, it is faster to gather and scatter the elements as half as many 64b elements. If the numbers are 64b, then it is usually faster to load and insert 128b values instead of doing a gather.

There are multiple HW prefetchers in the Intel® Xeon Phi™ processor tile. The one inside the core analyzes all the accesses in the data cache and the instructions that generated them - the Instruction Pointer Prefetcher (IPP). The prefetcher will then attempt to insert HW prefetches to the L1 cache if a strided access pattern is detected on a cacheable page. The IPP will not cross a 4k page boundary. The IPP uses the instruction address and thread to index into a table. For this reason, the compiler may insert NOPs into large loops (>256 B) to make instructions that access memory go into different table entries.

The L2 HW prefetcher tries to identify streaming access patterns, and can track up to 48 access patterns. A streaming access pattern touches consecutive cache lines in

increasing or decreasing order – the stride detected in the L2 is always +/-1 cacheline. The 48 detectors are allocated independently of the thread that originated the request. Each detector looks at the accesses done within a 4 KB region. If a stream is detected, HW prefetches for later elements of the stream will be sent to the L2 cache, and if they miss, to memory. The HW prefetcher will not stream across a 4 KB boundary. If multiple access patterns are done within the same 4 KB region, the detector can get confused, and fail to detect the stream.

The Intel® Xeon Phi™ processor supports many flavors of SW prefetch instructions. The Intel® Xeon Phi™ processor is more resilient to cache misses than Intel® Xeon Phi™ coprocessor, so programmers with experience from the previous iteration of Xeon Phi should not feel compelled to aggressively insert SW prefetches. With the two HW prefetchers described in previous paragraphs, most streaming and short stride access patterns should be detected by HW prefetchers. If the access pattern is streaming, then a programmer might benefit from SW prefetches beyond the current 4k page. If the access pattern is known, but non-streaming, then SW prefetches can be beneficial. This is especially true if the access pattern is a relatively large stride (>256 bytes), since the IPP will not fetch across a 4k boundary. The SW prefetch will do the PMH walk to fill the TLB, and to start the memory reference early.

Generally, SW prefetching into the L2 will show more benefit than L1 prefetches. L1 prefetches hold critical HW resources (fill buffer) until the cacheline fill completes. L2 prefetches do not hold those resources, and it is less likely that inserting an L2 SW prefetch will negatively impact performance. If you do use L1 SW prefetches, it is best if the SW prefetch hits in the L2 cache, so the length of time that the HW resources are held is minimized.

Adding useless SW prefetches can degrade performance. SW prefetches consume resources, just like any other instruction. If you use SW prefetches for addresses that are not valid, or mapped by the OS, you will experience a significant slowdown after inserting the SW prefetches. The performance monitoring event NUKE.ALL provides an indication of when this might be affecting your code.

The MEC is partially out of order. Memory accesses are dispatched from the scheduler in-order, but can complete in any order. This impacts performance negatively when the second oldest memory uop is ready to dispatch (address is good), but the oldest memory uop has not yet dispatched (memory address is not good). This can be easily observed in algorithms that do pointer chasing. If code loads the pointer from memory (cycle 0), and then de-references it in the next uop, the earliest point that uop can dispatch is in cycle 4. This means that no other MEC uops from that thread can dispatch in the second slot of cycle 0, or in any slot of cycles 1, 2, or 3. A common example of pointer chasing relevant to supercomputing is when the base address of many arrays (&a[0]) are kept on the stack. The compiler should try to maximize the number of memory operations between the load of the base address, and the instruction that dereferences it. A simple option is to load base pointers as consecutive memory references, and then dereference them in pairs. For instance, if you want to access a[i] and b[i] do this:

```
movq    r15, [rsp+0x40]      ;;; cycle N (load &a[0])
movq    r14, [rsp+0x48]      ;;; cycle N+1 (load &b[0])
vmovups zmm1, [r15+rax*8]    ;;; executes in cycle N+4
vmovups zmm2, [r14+rax*8]    ;;; cycle N+5
```

The naïve sequence, which interchanges the second and third instructions, is 3 cycles slower:

```
movq    r15, [rsp+0x40]      ;;; cycle N (load &a[0])
vmovups zmm1, [r15+rax*8]    ;;; executes in cycle N+4
movq    r14, [rsp+0x48]      ;;; cycle N+4 (load &b[0])
vmovups zmm2, [r14+rax*8]    ;;; cycle N+8
```

The advantage of the naïve sequence is that it only requires a single integer register to hold the base address. The faster sequence requires two integer registers for the base addresses, which can put more pressure on the register allocator. The benefit of the first sequence increases if the first pointer load misses the data cache.

If there are many loads in the machine, it might be possible to hoist up the pointer loads, so that there are several memory references between the pointer load and dereference, without requiring more integer registers to be reserved.

Store to load forwarding is simplistic on the Intel® Xeon Phi™ processor. Integer loads and stores (RBX, EAX) can forward as long as the store and load have the same memory address and the load is not larger than the store. Vector, x87, and MMX loads and stores can forward (ZMM0, YMM1, XMM2, MM3, & ST4) with the same conditions. VPU stores cannot forward to integer loads, and integer stores cannot forward to VPU loads. In either case, the load must wait until the store is post-retirement to get the value from memory.

Vector stores that use a mask (a k-register other than k0) cannot be forwarded from. If your algorithm requires such behavior, you may benefit if you merge the value in a register, and then store to memory using k0. Later loads can then forward from the merged value.

The memory hierarchy that caches lines and determines forwarding uses the address of the access. The L1 data cache uses bits 11:6 to identify which cache set to use. Forwarding logic uses bits 11:0 and the size of the access to identify potential forwarding or conflicts between loads and stores. If there are many conflicts, performance could be degraded. Unfortunately, many dynamic memory allocation routines (dependent on OS and compiler) will start large memory regions with the same bottom 12 bits. If your program accesses many arrays with identical shapes (element size and dimensions) and similar indices, performance could be significantly degraded. It is beneficial for bits [11..6] of memory accesses to be different. For example:

```
a = malloc(sizeof(double) * 10000);
b = malloc(sizeof(double) * 10000);
// very likely in most OSes that (a & 0xfff) == (b & 0xfff)
for (i=0; i < 10000; i++) {
     // a[i] and b[i] of iteration N collide
     // a[i] of iteration N-1 and
     // b[i-1] of iteration N collide
     a[i] = b[i] + 0.5 * b[i-1]);
}
```

There are multiple ways to offset dynamic arrays. If free() is not relevant, you can just allocate a few hundred extra bytes and manually offset the arrays: b = (double *)((char *)b + 192). You can also code your own versions of memory allocation routines to achieve this. Alternatively, you can use posix_memalign() with different alignment directives for each dynamic allocation to induce the OS to provide different memory alignments – like 64, 128, 256, or 512. Many scientific applications that use large arrays are sensitive to this. The SPEC06 application leslie3d can be affected by this quite easily.

When using VGATHER and VSCATTER, you often need to set a mask to all ones. An efficient instruction to do this is KXNOR of a mask register with itself. Since VSCATTER

and VGATHER clear their mask as the last thing they do, a loop carried dependence from the VGATHER to KXNOR can be generated. Because of this, it is wise to avoid using the same mask for source and destination in KXNOR. Since it is rare for the k0 mask to be used as a destination, it is likely that "KXNORW k1, k0, k0" will be faster than "KXNOR k1, k1, k1".

When writing memory that is unlikely to be accessed in the future (e.g. Streams Triad) the SW programmer has a choice between streaming stores and standard stores. If using a flat memory mode, streaming stores will likely perform better. If using cache mode, and the data being written fits in the MCDRAM cache, it is likely that standard stores will perform better. Experimenting with both options may yield non-trivial performance for your application.

7 Compiler Knobs & directives

For Fortran programmers using Fortran 90 syntax, please use the CONTIGUOUS attribute at appropriate points. If you do not, the compiler may assume that incoming arrays are not contiguous, and will (potentially) replace vector load and store instructions with VGATHER and VSCATTER instructions. This can have a negative impact on performance.

Expert coders compiling with the Intel compiler can annotate their code with various pragmas. Some of the more useful ones are loop_count, simd, and unroll. Please read the documentation for these pragmas, and use them where appropriate. The compiler can produce better code when it is given more information.

When using the Intel compilers, you can target the Intel® Xeon Phi™ processor with the –xMIC-AVX512 knob.

8 Numeric Sequences

Many AVX-512F sequences are in the AVX-512 optimization manual linked at the top of the document.

The Intel Compiler supports many Intel® Xeon Phi™ processor specific numeric sequences, taking advantage of AVX-512ER instructions. Compiling with a Intel® Xeon Phi™ processor target, you can generate many alternate sequences for division, square root, and exponentials by using various combinations of –no-prec-div, -no-prec-sqrt, -fp-model, and the -fimf-*** knobs. By trading off IEEE compatibility, you can get improved performance.

9 Code examples

In HPC computing, there are many kernels and accesses that are common. Below is the assembly code for several of these kernels, which show optimized code for the Intel® Xeon Phi™ processor.

Below is macrocode for a horizontal reduction of 32b elements. This is for single precision FP, but can be easily altered for a 32b integer reduction.

```
;; vector to reduce is in zmm6
vextractf64X4  zmm1, zmm6, 0x1
vaddps         ymm1, ymm6, ymm1
vpermpd        ymm4, ymm1, 0xFF
vpermpd        ymm5, ymm1, 0xAA
vpermpd        ymm3, ymm1, 0x44
vaddps         xmm1, xmm1, xmm4
vaddps         xmm3, xmm5, xmm3
vaddps         xmm3, xmm1, xmm3
vpsrlq         xmm1, xmm3, 32
vaddss         xmm3, xmm3, xmm1
```

This is the sample code for a 64b FP horizontal reduction:

```
;; vector to reduce is in zmm6
vextractf64X4 zmm1, zmm6, 0x01
vaddpd        ymm1, ymm6, ymm1
valignq       ymm4, ymm1, 0x3
valignq       ymm5, ymm1, 0x2
valignq       ymm3, ymm1, 0x1
vaddsd        ymm1, ymm1, ymm4
vaddsd        ymm3, ymm5, ymm3
vaddsd        ymm3, ymm1, ymm3
```

Here is a simplified code fragment for the inner loop of DGEMM, trying to compute C = A * B.

```
;; matrix - matrix dense multiplication
prefetcht0  [rdi+0x400] ;; get A matrix element into L1$
vmovapd     zmm30, [%rdi]
prefetcht0  [rsi+0x400] ;; get B matrix element into L1$
vfmadd231pd zmm1, [rsi+r12]{b}, zmm30  ;; b-cast B elem
vfmadd231pd zmm2, [rsi+r12+0x08]{b}, zmm30
vfmadd231pd zmm3, [rsi+r12+0x10]{b}, zmm30
vfmadd231pd zmm4, [rsi+r12+0x18]{b}, zmm30
vfmadd231pd zmm5, [rsi+r12+0x20]{b}, zmm30
vfmadd231pd zmm6, [rsi+r12+0x28]{b}, zmm30
vfmadd231pd zmm7, [rsi+r12+0x30]{b}, zmm30
vfmadd231pd zmm8, [rsi+r12+0x38]{b}, zmm30
prefetcht0  [rsi+0x440]      ;; pull line into the L1$
vfmadd231pd zmm9, [rsi+r12+0x40]{b}, zmm30
vfmadd231pd zmm10, [rsi+r12+0x48]{b}, zmm30
vfmadd231pd zmm11, [rsi+r12+0x50]{b}, zmm30
vfmadd231pd zmm12, [rsi+r12+0x58]{b}, zmm30
vfmadd231pd zmm13, [rsi+r12+0x60]{b}, zmm30
vfmadd231pd zmm14, [rsi+r12+0x68]{b}, zmm30
vfmadd231pd zmm15, [rsi+r12+0x70]{b}, zmm30
vfmadd231pd zmm16, [rsi+r12+0x78]{b}, zmm30
```

The code above has 16 partial sums. There should always be FMA instructions ready to execute in the VPU (6 cycles per FMA, up to 2 FMAs per cycle). It is important to keep the average instruction length at 8 bytes or less, to enable maximum throughput. This is why the index register (r12) is used, so the displacement used in the FMAs can be kept small. At the end of the inner loop, the partial sums will need to be added to produce a single value to be stored out (to the C matrix).

Here is an efficient way to compare 2 YMM registers with 32b values and get the result in a k-mask register:

```
vpcmpd      ymm1, ymm2, ymm3
vptestmd    k1, zmm1, zmm1
```

Here is an efficient way to get YMM masks into a k-mask register:

```
vptestmd    k1, zmm1, zmm1
```

This is how to get a k-mask into the YMM mask format:

```
VPTERNLOGQ  zmm0 {k1}{z}, zmm0, zmm0, 0xFF
```

A compiler can replace VCOMIS* instructions (or similar versions) with a 2 instruction sequence like this:

```
VCMPSS      k0, zmm1, zmm2, IMM  ;; Change IMM for comparison
KORTEST     k0, k0               ;; Set EFLAGs
```

10 Direct mapped MCDRAM cache

Using cache memory mode, the MCDRAM cache is a convenient way to increase memory BW. As a memory side cache, it can automatically cache recently used data, and provide much higher BW than what DDR memory can achieve. When MCDRAM is placed in cache mode, it is a direct mapped cache. This means that multiple memory locations map to a single place in the cache. Because of this, a simple first optimization for a program is to turn on the MCDRAM cache. Some applications that heavily utilize a few GB of memory could see performance improvements of up to 4x. Because of the simplicity of this – no source code changes, and the large possible performance benefits, moving from DDR only to MCDRAM cache mode should be one of the first performance optimizations to try.

There are a few scenarios where enabling the cache could reduce performance. One case is when the MCDRAM cache is not able to hold the accessed working set. If an application streams through 64 GB of memory without reuse, then checking the MCDRAM cache (and missing) will only increase latency to DDR memory.

Another thing to note is that the MCDRAM direct mapped cache uses the physical address, not the linear address. Even if an address is contiguous in the linear / virtual address, the physical addresses that the OS gives to the application memory are not required to be. This can cause cache contention when using a significant portion of the MCDRAM cache. This is likely to reduce the peak memory BW achievable. This can vary from run to run, as how the OS allocates pages can change from run to run. Using performance monitoring HW to compute the MCDRAM cache hit rate can be instructive in diagnosing this (UNC_E_EDC_ACCESS). Many benchmarks have shown the run to run variation is "in the noise".

If MCDRAM cache is enabled, every modified line in the tile caches (L1 or L2 cache) must have an entry in the MCDRAM cache. If a line is evicted from the MCDRAM cache, then any modified version in the tile caches will writeback its data to memory and transition to a clean state. There is a very small probability that a pair of lines that are frequently read and written will map to the same MCDRAM set. This could cause a pair of writes that would normally hit in the L1 caches to become writes that need to go to DDR when MCDRAM cache mode is enabled. This would cause a pair of threads to become substantially slower than the other threads in the chip. Linear to physical mapping can vary from run to run, making it difficult to diagnose.

The instances that we have observed this behavior, have been when two threads read and write their private stacks. Conceptually, any data location that is commonly read and written would work, but register spills to the stack are the most frequent case. If the stacks are offset by a multiple of 16 GB (or the total MCDRAM cache size) in physical memory, they would collide into the same MCDRAM cache set. A run-time that forced all thread stacks to allocate into a contiguous HW memory region would avoid this case from occurring.

There is HW to reduce the frequency of set conflicts from occurring. The probability of hitting this scenario on a given node is extremely small. The best clue to detecting this, is that a pair of threads on the same chip are significantly slower than all other threads during a program phase. The threads, cores, and packages that collide should vary from run to run, happen rarely, and only when the cache memory mode is enabled. It is very unlikely that a user ever encounters this on their system.

11 Scalar versus Vector code

When trying to generate assembly code for a loop, the compiler or programmer might know that the loop is vectorize-able, and have the choice between scalar and vector code. Which method is best for performance?

A very simple guideline is that loops with trip counts larger than 16 will be faster with vector code, and scalar code is faster for trip counts less than 4. In between those two values, vector and scalar code has about the same performance.

For the rest of this section, a slightly more sophisticated algorithm to choose between scalar and vector code is discussed. It assumes that the operations in the loop and the number of iterations the loop will execute are known.

Below, the concept of "horizontal reduction" is discussed. A horizontal reduction operation transforms a vector into a scalar value. These generally occur at the end of a loop when a scalar value must be produced from the vector registers. The most common type of horizontal reduction sums the vector elements.

Table 4. Operation Types

Operation	Cost	Example
Simple math	1	A*B+C
Load (splits cacheline)	1 (2)	A[i]
Store (splits cacheline)	1 (2)	A[i] = 2;
Gather (scatter) 8 element	15 (20)	A[column[i]]
Gather (scatter) 16 element	20 (25)	A[column[i]]
Horizontal reduction	30	sum += A[i]
Division or square root	15	A/B

It is easiest to explain the algorithm via examples. Due to differences in various compilers' internal formats and programmers' knowledge, the algorithm has to be a bit flexible. Some of the corner cases are glossed over.

The following is a simple horizontal reduction loop:

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

There are 2 loads (a[i] & b[i]), an FMA, and a horizontal reduction in the vector version (to get sum). If compiled to scalar code, the horizontal reduction becomes a scalar add per loop iteration.

The heuristic costs are 4*N for scalar code, and 3*ceiling(N/8) +30 for DP AVX-512 vector code. The vector version of the code assumes that the main loop and remainder loop are vectorized. The break-even point between scalar and vector code would be around N=9. If N is smaller, then scalar is better. If N is larger, then vectorized code will be faster.

Another example, this time using gathers:

```
for (i=0; i<N; i++) {c[i] = a[indir[i]] * K + b[i]; }
```

There are 2 loads (indir[i] & b[i]), an FMA, a store (c[i] =), and a gather / load, depending on whether vector or scalar code is produced. The costs are 5*N for the scalar code and 19*ceiling(N/8) for the vector code. Scalar is better if N < 4.

This example uses more gathers:
```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

There is one load (ind[i]), an FMA, a store (c[i] =) and 2 gathers / loads. The scalar cost is 5*N, and the vector cost is 33*ceiling(N/8). Scalar is better for small values of N. For very large values of N, vector code is probably better, but the margin is likely small.

This example has a gather and a horizontal reduction:
```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

There are 2 loads (ind[i] & b[i]), an FMA, a gather/load, and a horizontal reduction/sum. The scalar cost is 5*N, and the vector cost is 19*ceiling(N/8) + 30. Scalar code is better for N<=13, and vector code is probably better for larger trip counts.

This example has a scatter and division:
```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

There are 3 loads (a[i], b[i], & ind[i]), a scatter/store, and a division (a[]/b[]). The scalar cost is 19*N, and the vector cost is 38*ceiling(N/8). Scalar code is better if N <= 1.

The next example has a gather and scatter operation:
```
for (i=0; i<N; i++) {b[ind[i]] = a[ind[i]]; }
```

There is one load (ind[i]), a scatter/store (b[ind[i]]), and a gather/load (a[ind[i]]). The scalar cost is 3*N, and the vector cost is 36*ceiling(N/8). According to the heuristic, scalar code is always better. With the in-order MEC, it is likely that the scalar version of the loop would benefit from being unrolled a bit.

The last example is a code fragment from miniMD:
```
for (int k = 0; k < numneigh; k++) {
       int j = neighs[k];
       double rsq = (xtmp - x[3*j])^2 +
                 (ytmp - x[3*j+1])^2 +
                 (ztmp - x[3*j+2])^2;
       if (rsq < cutforcesq) {
            double sr2 = 1.0/rsq;
            double sr6 = sr2*sr2*sr2;
            double force = sr6*(sr6-0.5)*sr2;
            res1 += delx*force;
            res2 += dely*force;
            res3 += delz*force;
       }
}
```

Outside the IF clause, there is a load, 3 gathers, and 6 math operations. Inside the IF clause, there is a division, 8 math operations, and 3 horizontal reductions. The scalar cost is 10*numneigh + 23 * numneigh * percent_rsq_less_than_cutforcesq. The vector cost is (52+23) * ceiling(numneigh / 8) + 3 * 30. Scalar code makes sense if numneigh < 6 or if the compiler is highly confident that the if clause is almost never taken.

For many compilers, a vectorized loop is generated, and a remainder loop is used to take care of the rest of the operations. In other words, the vectorized loop is executed floor(N/8) times, and the remainder loop is executed N mod 8 times. In that case, modify the equations above to use floor instead of ceiling to determine whether the primary loop should be vectorized. For the remainder loop, the maximum value of the loop trip count is known. If N is unknown, it is simplest to set N to half the maximum value (4 for a ZMM vector of doubles).

More sophisticated analysis can be done in this area. This is a starting point for optimizers.

§